

Dynamic Range Majority Data Structures^{*}

Amr Elmasry¹, Meng He², J. Ian Munro³, and Patrick K. Nicholson³

¹ Department of Computer Science, University of Copenhagen, Denmark

² Faculty of Computer Science, Dalhousie University, Canada

³ David R. Cheriton School of Computer Science, University of Waterloo, Canada,
elmasry@diku.dk, mhe@cs.dal.ca, {imunro, p3nichol}@uwaterloo.ca

Abstract. Given a set \mathcal{P} of n coloured points on the real line, we study the problem of answering range α -majority (or “heavy hitter”) queries on \mathcal{P} . More specifically, for a query range \mathcal{Q} , we want to return each colour that is assigned to more than an α -fraction of the points contained in \mathcal{Q} . We present a new data structure for answering range α -majority queries on a dynamic set of points, where $\alpha \in (0, 1)$. Our data structure uses $O(n)$ space, supports queries in $O((\lg n)/\alpha)$ time, and updates in $O((\lg n)/\alpha)$ amortized time. If the coordinates of the points are integers, then the query time can be improved to $O(\lg n/(\alpha \lg \lg n))$. For constant values of α , this improved query time matches an existing lower bound, for any data structure with polylogarithmic update time. We also generalize our data structure to handle sets of points in d -dimensions, for $d \geq 2$, as well as dynamic arrays, in which each entry is a colour.

1 Introduction

Many problems in computational geometry deal with point sets that have information encoded as colours assigned to the points. In this paper, we design dynamic data structures for the *range α -majority problem*, in which we want to report colours that appear *frequently* within an axis-aligned query rectangle. This problem is useful in database applications in which we would like to know typical attributes of the data points in a query range [23,24]. For the one-dimensional case, where the points represent time stamps, this problem has data mining applications for network traffic logs, similar to those of coloured range counting (cf. [17]).

Formally, we are given a set, \mathcal{P} , of n points, where each point $p \in \mathcal{P}$ is assigned a colour c from a set, C , of colours. We denote the colour of p as $\text{col}(p) = c$. We are also given a fixed parameter $\alpha \in (0, 1)$, that defines the threshold for determining whether a colour is to be considered frequent. Our goal is to design a *dynamic range α -majority data structure* that can perform the following operations:

- QUERY(\mathcal{Q}): We are given an axis-aligned hyperrectangle \mathcal{Q} as a query. Let $\mathcal{P}(\mathcal{Q})$ be the set $\{p \mid p \in \mathcal{P} \cap \mathcal{Q}\}$, and $\mathcal{P}(\mathcal{Q}, c)$ be the set $\{p \mid p \in \mathcal{P}(\mathcal{Q}), \text{col}(p) = c\}$. The answer to the query \mathcal{Q} is the set of colours C^* such that for each colour $c \in C^*$, $|\mathcal{P}(\mathcal{Q}, c)| > \alpha |\mathcal{P}(\mathcal{Q})|$, and for all $c \notin C^*$, $|\mathcal{P}(\mathcal{Q}, c)| \leq \alpha |\mathcal{P}(\mathcal{Q})|$. We refer to a colour $c \in C^*$ as an *α -majority* for \mathcal{Q} , and this type of query as an *α -majority query*. When $\alpha = 1/2$, the problem is to identify the majority colour in \mathcal{Q} , if such a colour exists.
- INSERT(p, c): Insert a point p with colour c into \mathcal{P} .
- DELETE(p): Remove the point p from \mathcal{P} .

1.1 Previous Work

Static and Dynamic Range α -Majority: In all of the following results, unless mentioned otherwise, the threshold $\alpha \in (0, 1)$ is fixed at construction time, rather than specified for each query individually.

^{*} A preliminary version of this work appeared in the 22nd International Symposium on Algorithms and Computation (ISAAC 2011). This work was supported by NSERC of Canada, and NSERC PGS-D Scholarship, and the Canada Research Chairs Program.

Karpinski and Nekrich [23] studied the problem of answering range α -majority queries, which they call *coloured α -domination* queries. In the static case, they gave an $O(n/\alpha)$ space data structure that supports one-dimensional queries in $O((\lg n \lg \lg n)/\alpha)$ time⁴, and an $O((n \lg \lg n)/\alpha)$ space data structure that supports queries in $O((\lg n)/\alpha)$ time. In the dynamic case, they gave an $O(n/\alpha)$ space data structure for one-dimensional queries that supports queries and insertions in $O((\lg^2 n)/\alpha)$ time, and deletions in $O((\lg^2 n)/\alpha)$ amortized time. They also gave an alternative $O((n \lg n)/\alpha)$ space data structure that supports queries and insertions in $O((\lg n)/\alpha)$ time, and deletions in $O((\lg n)/\alpha)$ amortized time. For points in d -dimensions, for constant $d \geq 2$, they gave a static $O((n \lg^{d-1} n)/\alpha)$ space data structure that supports queries in $O((\lg^d n)/\alpha)$ time, as well as a dynamic $O((n \lg^{d-1} n)/\alpha)$ space data structure that supports queries and insertions in $O((\lg^{d+1} n)/\alpha)$ time, and deletions in $O((\lg^{d+1} n)/\alpha)$ amortized time.

Durocher *et al.* [12] described a static $O(n(\lg(1/\alpha) + 1))$ space data structure that answers range α -majority queries in an array in $O(1/\alpha)$ time. This data structure is based on the idea that it is possible to produce a short list of candidate α -majorities for any query, and then efficiently verify the frequencies of these candidates using succinct data structures. In a later version of the same paper [13], they described how to extend their technique to d -dimensions for constant $d \geq 2$, resulting in an $O(n \lg^{d-1} n)$ space data structure that supports range α -majority queries in $O(\lg^d n/\alpha)$ time. Gagie *et al.* [16] improved the static one-dimensional result to $O(n(\min(\lg(1/\alpha), H) + 1))$ space, where $H \leq \lg n$ is the 0th-order empirical entropy of the sequence stored in the array. The same authors also described how to improve the query time to $O(1/\beta)$, when asked for the β -majorities in a query range, for any $\beta \geq \alpha$ specified at query time. Recently, for the two-dimensional static case, Wilkinson [31] presented an improved data structure that occupies $O(n \lg^\varepsilon n \lg(1/\alpha))$ space, for any constant $\varepsilon > 0$, and can answer queries in $O(\lg n/\alpha)$ time.

Approximate Versions of the Problem: Researchers have also examined an approximate version of the range α -majority problem, in which the solution must contain all the α -majorities in a query range, but can also contain some false positives. Lai *et al.* [24] studied the dynamic problem, using the term *heavy-colours* instead of α -majorities. They presented a dynamic data structure based on sketching, which provides an approximate solution with probabilistic guarantees for constant values of α . For one dimension their data structure uses $O(hn)$ space and supports queries and updates in $O(h \lg n)$ time, where the parameter $h = O(\frac{\lg |C|}{\varepsilon} \lg(\frac{\lg |C|}{\alpha \delta}))$ depends on the threshold α , the approximation factor ε , the total number of colours $|C|$, and the probability of failure δ . They also noted that the space can be reduced to $O(n)$, at the cost of increasing the query time to $O(h \lg n + h^2)$. Thus, for constant values of ε , δ , and α , their data structure uses $O(n)$ space and has $O((\lg n \lg \lg n)^2)$ query and update time in the worst case when $\lg m = \Omega(\lg n)$.

Another approximate data structure based on sketching was proposed by Wei and Yi [30]. Their data structure uses linear space, answers queries in $O(\lg n + 1/\varepsilon)$ time, and may return false positives with relative frequency between $\alpha - \varepsilon$ and α . The cost of updates is $O(\mu \lg n \lg(1/\varepsilon))$ amortized time, where μ is the cost of updating the sketches. We note that this result was obtained independently of ours, and that both our techniques and the main technique they develop, called *exponential decomposability*, are similar. By combining Theorem 4 of their paper with standard range counting data structures, it is not difficult to get a data structure that occupies linear space, answers queries in $O(\lg n/\alpha)$ time, and supports updates in $O((\lg n \lg(1/\alpha))/\alpha)$ amortized time for the non-approximate version of the problem that we study. However, we slightly improve this update time, and also generalize our data structures to higher dimensions, whereas their structure is part of a more general framework that supports other kinds of aggregate queries.

Lower Bounds: The partial sum problem for threshold functions [21] captures the essence of the dynamic range α -majority problem: maintain n bits x_1, \dots, x_n subject to updates and *threshold queries*. An update consists of flipping the bit at a specified index. The answer to query **threshold**(i) is “yes” if and only if $\sum_{j=1}^i x_j \geq f(i)$, where $f(i)$ is an integer function such that $f(i) \in \{0, \dots, \lceil i/2 \rceil\}$. Husfeldt and Rauhe proved a lower bound [21] on the query time t_q for a data structure that can answer threshold queries with update time t_u .

⁴ We use $\lg n$ to denote $\lceil \log_2 n \rceil$.

Any data structure for dynamic α -majority can be used to solve the partial sum problem for threshold functions. In particular, we can treat the problem as involving n points with integer coordinates $1, \dots, n$, with each point having one of two colours. A flip operation can be implemented as a deletion followed by an insertion. Thus, we can state their lower bound in terms of our problem, denoting the cell size of our machine as w :

Lemma 1 (Follows from [21], Prop. 4). *Let t_u and t_q denote the update and query times, respectively, for any dynamic α -majority data structure. Then,*

$$t_q = \Omega \left(\frac{\lg(\min\{\alpha n, (1 - \alpha)n\})}{\lg(t_u w \lg(\min\{\alpha n, (1 - \alpha)n\}))} \right).$$

This bound suggests that, for constant values of α and word size $\Theta(\lg n)$ bits, $O(\lg n / \lg \lg n)$ query time for integer point sets is optimal for any data structure with polylogarithmic update time.

Other Related Work: Finally, several other results exist for finding α -majorities in the streaming model, typically referred to as *heavy hitters* [6,10,22,25]. De Berg and Haverkort [9] studied a similar problem of reporting τ -significant colours. For this problem, the goal is to output all colours c such that at least a τ -fraction of *all* of the points with colour c lie in the axis-aligned query rectangle. More broadly, there are other data structure problems that deal with coloured points. In *coloured range reporting problems*, we are interested in reporting the set of distinct colours assigned to the points contained in an axis-aligned rectangle. Similarly, in the *coloured range counting problem* we are interested in returning the number of such distinct colours. Gupta *et al.* [20], Bozanis *et al.* [7], and, more recently, Gagie *et al.* [18] and Gagie and Kärkkäinen [17] studied these problems and presented several interesting results.

1.2 Our Results

In this paper we present new data structures for the dynamic range α -majority problem in the word-RAM model with word size $\Omega(\lg n)$, where n is the number of points in the set \mathcal{P} , and $\alpha \in (0, 1)$. Our results are summarized and compared to the previous best results in Table 1. The *input* column indicates the type of data we are considering. We use *points* to denote a set of points on a line with real-valued coordinates that we can compare in constant time, *integers* to denote a set of points on a line with word sized integer coordinates, and *array* to denote that the input is to be considered a dynamic array, where the positions of the points are in rank space.

Source	Input	Space	Query	Insert	Delete
[23, Thm. 3]	points	$O(\frac{n}{\alpha})$	$O(\frac{\lg^2 n}{\alpha})$	$O(\frac{\lg^2 n}{\alpha})$	$O(\frac{\lg^2 n}{\alpha})^*$
[23, Thm. 3]	points	$O(\frac{n \lg n}{\alpha})$	$O(\frac{\lg n}{\alpha})$	$O(\frac{\lg n}{\alpha})$	$O(\frac{\lg n}{\alpha})^*$
New	points	$O(n)$	$O(\frac{\lg n}{\alpha})$	$O(\frac{\lg n}{\alpha})^*$	$O(\frac{\lg n}{\alpha})^*$
New	integers	$O(n)$	$O(\frac{\lg n}{\alpha \lg \lg n})$	$O(\frac{\lg n}{\alpha})^*$	$O(\frac{\lg n}{\alpha})^*$
New	array	$O(n)$	$O(\frac{\lg n}{\alpha \lg \lg n})$	$O(\frac{\lg^3 n}{\alpha \lg \lg n})^*$	$O(\frac{\lg n}{\alpha})^*$

Table 1. Comparison of the results in this paper to the previous best results. For the entries marked with “*” the running times are amortized.

Our results improve upon previous results in several ways. Most noticeably, all our data structures require linear space. In order to provide fast query and update times for our linear space structures, we prove several interesting properties of α -majority colours. We note that the lower bound from Lemma 1 implies that, for constant values of α , an $O(\lg n / \lg \lg n)$ query time for integer point sets is optimal for any data structure

with polylogarithmic update time, when the word size $w = \Theta(\lg n)$. Our data structure for points on a line with integer coordinates achieves this optimal query time.

Our data structures can also be generalized to handle d -dimensional points, improving upon previous results in the dynamic case [23]. For $d \geq 2$, our data structure occupies $O(n \lg^{d-1} n)$ space, answers range α -majority queries in $O((\lg^d n)/\alpha)$ time, and supports updates in $O((\lg^d n)/\alpha)$ amortized time.

Road Map: In Section 2 we present a dynamic range α -majority data structure for points in one dimension. In Section 3 we show how to speed up the query time of our data structure in the case where the points have integer coordinates. In Section 4 we generalize our one dimensional data structures to higher dimensions. Finally, in Section 5 we present our data structure for dynamic arrays.

Assumptions About Colours: In the following sections, we assume that we can compare colours in constant time. In order to support a dynamic set of colours, we employ the techniques described by Gupta *et al.* [20]. These techniques allow us to maintain a mapping from the set of colours to integers in the range $[1, 2n]$, where n is the number of points *currently* in our data structure. This allows us to index into an array using a colour in constant time.

For the dynamic problems discussed, this mapping is maintained using a method similar to global rebuilding to ensure that the integer identifiers of the colours do not grow too large [20, Section 2.3]. When a coloured point is inserted, we must first determine whether we have already assigned an integer to that colour. By storing the set of known colours in a balanced binary search tree, this can be checked in $O(\lg |C|)$ time; recall that $|C|$ is the number of distinct colours currently assigned to points in our data structure. Since $|C| \leq n$, this cost is absorbed by update time of our data structure; see Table 1. Therefore, from this point on, we assume that we are dealing with integers in the range $[1, 2n]$ when we discuss colours.

2 Dynamic Data Structures in One Dimension

In one-dimension we can interchange the notion of points and x -coordinates in \mathcal{P} , since they are equivalent. Depending on the context we may use either term. Our basic data structure, like that of Karpinski and Nekrich [23], is a modified weight balanced B-tree [3]. However, we prove several interesting combinatorial properties of α -majorities in order to provide more efficient support for queries and updates.

Let T be a weight-balanced B-tree with branching parameter 8 and leaf parameter 1 such that each leaf represents an x -coordinate in \mathcal{P} . From left to right the leaves are sorted in ascending order of the x -coordinate that they represent. Let $T(u)$ be the subtree rooted at node u . Each internal node u in the tree represents a range $R(u) = [x_{\min}, x_{\max}]$, where x_{\min} is the x -coordinate represented by the leftmost leaf in $T(u)$, and x_{\max} is the x -coordinate represented by the rightmost leaf in $T(u)$. We number the levels of the tree $0, 1, \dots, \Theta(\lg n)$ from top to bottom. If a node is h levels above the leaf level, we say that this node is of *height* h . By the properties of weight-balanced B-trees, the range represented by an internal node of height h contains at least $8^h/2$ (except the root) and at most 2×8^h points, and the degree of each internal node is at least 2 and at most 32.

2.1 Supporting Queries

Given a query $\mathcal{Q}' = [x'_a, x'_b]$, we perform a top-down traversal on T to map \mathcal{Q}' to the range $\mathcal{Q} = [x_a, x_b]$, where x_a and x_b are the points in \mathcal{P} with x -coordinates that are the successor and the predecessor of x'_a and x'_b , respectively. We call the query range \mathcal{Q} *general* if \mathcal{Q} is not represented by a single node of T . We first define the notion of *representing* a general query range by a set of nodes:

Definition 1. *Given a general query range $\mathcal{Q} = [x_a, x_b]$, \mathcal{Q} induces a set, I , of nodes in the tree T , satisfying the following two conditions.*

1. *The range represented by the parent of each node in I is not entirely contained in \mathcal{Q} .*
2. *For all $p \in \mathcal{P} \cap \mathcal{Q}$, there exists some node $u \in I$ with $p \in R(u)$.*

We say that I is the set of nodes in the tree T representing \mathcal{Q} .

For each node $u \in T$, we keep a list, $L(u)$, of k candidate colours, i.e., the k most frequent colours in the range $R(u)$ represented by u , breaking ties arbitrarily. Later, we will fix a value for k . Let $L^* = \cup_{u \in I} L(u)$, i.e., the union of all the candidate lists among the nodes representing the query range \mathcal{Q} . For each colour $c \in C$, we keep a separate range counting data structure, F_c , containing all points $p \in \mathcal{P}$ with colour c , and also a range counting data structure, F , containing all of the points in \mathcal{P} . Let m be the total number of points in the range $[x_a, x_b]$, which can be determined by querying F . For each $c \in L^*$, we query F_c with the range $[x_a, x_b]$ letting occ be the result. If $\text{occ} > \alpha m$, then we report that $c \in C^*$.

It is clear that I contains at most $\Theta(\lg n)$ nodes. Furthermore, if a colour c is an α -majority for \mathcal{Q} , then it must be an α -majority for at least one of the ranges in I [23, Observation 1]. If we set $k = \lceil 1/\alpha \rceil$ and store $\lceil 1/\alpha \rceil$ colours in each internal node as candidate colours, then, by the procedure just described, we will perform a range counting query on $\Theta((\lg n)/\alpha)$ colours. If we use balanced search trees for our range counting data structures, then this takes $\Theta((\lg^2 n)/\alpha)$ time overall. However, in the sequel we show how to improve this query time by exploiting the fact that the nodes in I that are closer to the root of T contain more points in the ranges that they represent.

We shall prove useful properties of a general query range \mathcal{Q} and the set, I , of nodes representing it in Lemmas 2, 3, 4, and 5. In these lemmas, m denotes the number of points in \mathcal{Q} , and i_1, i_2, \dots denote the distinct values of the heights of the nodes in I , where $i_1 > i_2 > \dots \geq 0$. We first give an upper bound on the number of points contained in the ranges represented by the nodes of I of a given height:

Lemma 2. *The total number of points in the ranges represented by all the nodes in I of height i_j is less than $m \times \min(1, 31 \times 8^{1-j})$.*

Proof. Since \mathcal{Q} is general and contains at least one node of height i_1 , m is greater than the minimum number of points that can be contained in a node of height i_1 , which is $8^{i_1}/2$. The nodes of I whose height is i_j , $j \neq 1$, are siblings and must have at least one sibling that is not in I . The number of points contained in the interval represented by this sibling is at least $8^{i_j}/2$. Therefore, the number, m_j , of points in the ranges represented by the nodes of I at level i_j is at most $2 \times 8^{i_j+1} - 8^{i_j}/2 = (31/2) \times 8^{i_j}$. Thus, $m_j/m < 31 \times 8^{i_j-i_1} < 31 \times 8^{1-j}$. \square

We next use the above lemma to bound the number of points whose colours are not among the candidate colours stored in the corresponding nodes in I .

Lemma 3. *Suppose we are given a node $v \in I$ of height i_j and a colour c . Let $n_v^{(c)}$ denote the number of points with colour c in $R(v)$, the range covered by v , if c is not among the first $k_j = \lceil k/2^{j-1} \rceil$ most frequent candidate colours in the candidacy list of v , and $n_v^{(c)} = 0$ otherwise. Then $\sum_{v \in I} n_v^{(c)} < \frac{5.59m}{k+1}$.*

Proof. If c is not among the first k_j candidate colours stored in v , then the number of points with colour c in $R(v)$ is at most $1/(k_j + 1)$ times the number of points in $R(v)$. Thus,

$$\begin{aligned} \sum_{v \in I} n_v^{(c)} &< \sum_{j=1}^2 \frac{m}{k_j + 1} + \sum_{j \geq 3} \frac{(31 \times 8^{1-j}) m}{k_j + 1} \\ &< \frac{m}{k+1} \left(1 + 2 + 31 \left(\frac{2^2}{8^2} + \frac{2^3}{8^3} + \dots \right) \right) \\ &< \frac{5.59m}{k+1} \end{aligned}$$

\square

We next consider the nodes in I that are closer to the leaf levels. Let I_t denote the nodes in I that are at one of the top $t = \lceil \frac{\lg(\frac{1}{\alpha})}{3} + 2.05 \rceil$ —not necessarily consecutive—levels of the nodes in I . We prove the following property:

Lemma 4. *The number of points contained in the ranges represented by the nodes in $I \setminus I_t$ is less than $\alpha m/2$.*

Proof. By Lemma 2, the number of points contained in the ranges represented by the nodes in $I \setminus I_t$ is less than:

$$\begin{aligned} 31m \sum_{j \geq t+1} 8^{1-j} &< 31m \left(\frac{1}{8^t} + \frac{1}{8^{t+1}} + \dots \right) \\ &< 31m \left(\frac{8}{7} \times \frac{1}{8^t} \right) \end{aligned}$$

Since $t \geq \frac{\lg(\frac{1}{\alpha})}{3} + 2.05$, the above value is less than $\alpha m/2$. \square

With the above lemmas, we can choose an appropriate value for k to guarantee the following property that is critical to achieve improved query time:

Lemma 5. *When $k = \lceil \frac{11.18}{\alpha} \rceil - 1$, any α -majority colour, c , of the query range \mathcal{Q} is among the union of the first $\lceil k/2^{j-1} \rceil$ candidates stored in each node of height i_j representing a range in I_t .*

Proof. The total number of points with colour c in the ranges represented by the nodes in $I \setminus I_t$ is less than $\alpha m/2$ by Lemma 4. By Lemma 3 and our choice for the value of k , less than $\alpha m/2$ points in the ranges represented by the nodes in I_t for which c is not a candidate can have colour c . The lemma thus follows. \square

For each node $v \in T$, we keep a semi-ordered list of the k candidate colours in the range $R(v)$ represented by v . The order on the colours for any candidacy list is maintained such that the most frequent $\lceil k/2^{j-1} \rceil$ colours come first, for all $j = 2, 3, \dots$, arbitrarily ordered within their positions. Note that such a semi-ordering can be obtained in $O(k)$ time by repeated median queries. That is, by using a linear time median finding algorithm [5], we can partition the list so that the first half of the list contains the $k/2$ most frequent colours, and then recurse on the first half of the list until the list has 1 element. In total, this takes $O(k + k/2 + \dots k/4) = O(k)$ time.

By setting $k = \lceil 11.18/\alpha \rceil - 1$, Lemma 5 implies that the colours that we have checked are the only possible α -majority colours for the query. Furthermore, Lemma 4 implies that we need only check the nodes on the top $O(\lg(1/\alpha))$ levels in I . Let I_t denote the set of nodes in these levels. We present the following lemma:

Lemma 6. *The data structures described in this section occupy $O(n)$ words, and can be used to answer a range α -majority query in $O((\lg n)/\alpha)$ time with the help of an additional array of size $2n$.*

Proof. To support α -majority queries, we only consider the nontrivial case in which the query range \mathcal{Q} is general. By Lemma 5, the α -majorities can be found by examining the first $\lceil k/2^{j-1} \rceil$ candidate colours stored in each node representing a range in I_t . Thus, there are at most $O(\lceil \frac{1}{\alpha} \rceil + \lceil \frac{1}{2\alpha} \rceil + \lceil \frac{1}{4\alpha} \rceil + \dots + \lceil \frac{1}{2^{t-1}\alpha} \rceil) = O(\frac{1}{\alpha})$ relevant colours to check. Let L_t denote the set of these colours. For each $c \in L_t$ we query our range counting data structures F_c and F in $\Theta(\lg n)$ time to determine whether c is an α -majority. Thus, the overall query time is $O((\lg n)/\alpha)$.

There are $\Theta(n)$ nodes in the weight-balanced B-tree. Therefore, one would expect the space to be $\Theta(n/\alpha)$ words, since each node stores $\Theta(1/\alpha)$ colours. We use a pruning technique on the lower levels of the tree in order to reduce the space to $O(n)$ words overall. If a node u covers less than $1/\alpha$ points, then we need not store $L(u)$, since every colour in $T(u)$ is an α -majority for $R(u)$. Instead, during a query, we can traverse the leaves of $T(u)$ in order to determine the unique colours. To make this efficient, we require an array D of size $2n$ integers to count the frequencies of the colours in $R(u)$. As mentioned in Section 1.2, we can map a colour into an index of the array D , which allows us to increment a frequency counter in $O(1)$ time. Thus, we can extract the unique colours in $R(u)$ in $O(|T(u)|) = O(\frac{1}{\alpha})$ time. The number of tree nodes whose subtrees have at least $1/\alpha$ leaves is $O(n\alpha)$. Thus, we store $O(k) = O(1/\alpha)$ words in $O(n\alpha)$ nodes, and the total space used by our B-tree T is $O(n)$ words. The only other data structures we make use of are the array D and the range counting data structures F and F_c for each $c \in C$, and together these occupy $O(n)$ words. \square

2.2 Supporting Updates

We next establish how much time is required to maintain the list $L(v)$ in node v under insertions and deletions. We begin by observing that it is not possible to lazily maintain the list of the top $k = \lceil \frac{11.18}{\alpha} \rceil - 1$ most frequent colours in each range: many of these colours could have low frequencies, and the list $L(v)$ would have to be rebuilt after very few insertions or deletions. To circumvent this problem, we relax our requirements on what is stored in $L(v)$, only guaranteeing that *all* of the β -majorities of the range $R(v)$ must be present in $L(v)$, where $\beta = \lceil \frac{11.18}{\alpha} \rceil^{-1}$. With this alteration, we can still make use of the lemmas from the previous section, since they depend only on the fact that there are no colours $c \notin L(v)$ with frequency greater than $\beta|T(v)|$. The issue now is how to maintain the β -majorities of $R(v)$ during insertions and deletions of colours.

Karpinski and Nekrich noted that if we store the $(\beta/2)$ -majorities for each node v in T , then it is only after $|T(v)|\beta/2$ deletions that we must rebuild $L(v)$ [23]. For the case of insertions and deletions, their data structure performs a range counting query at each node v along the path from the root of T to the leaf representing the inserted or deleted colour c . This counting query is used to determine if the colour c should be added to, or removed from, the list $L(v)$.

In contrast, our strategy is to be lazy during insertions and deletions, waiting as long as possible before recomputing $L(v)$, and to avoid performing range counting queries for each node in the update path. We provide a tighter analysis (to constant factors) of how many insertions and deletions can occur before the list $L(v)$ is to be rebuilt. One caveat is that the results in this section only apply when $\beta \in (0, \frac{1}{2}]$. However, since $\alpha < 1$, our choice of β satisfies this condition.

We use \mathbb{Z}^* to denote $\mathbb{Z}^+ \cup \{0\}$. The following lemma is used to show a lower bound on the number of update operations (insertions and deletions) that can occur before a list needs to be recomputed:

Lemma 7. *Let $\Gamma(\ell, j, \beta) = \min_{n_i \in \mathbb{Z}^*, n_d \in \mathbb{Z}^*} \left\{ n_i + n_d \mid \frac{j+n_i}{\ell+n_i-n_d} > \beta \right\}$ where $\ell \in \mathbb{Z}^+$, $j \in \mathbb{Z}^*$, $j < \ell$, and $\beta \in (0, \frac{1}{2}]$. If $\ell \geq 2j + 1$, then $\Gamma(\ell, j, \beta) \geq \frac{\beta\ell-j}{1-\beta}$.*

Proof. Observe that $\frac{j+1}{\ell+1} \geq \frac{j}{\ell-1}$ if and only if $\ell \geq 2j + 1$. This implies that increasing n_i rather than n_d by the same amount increases the value of the ratio $\frac{j+n_i}{\ell+n_i-n_d}$ by a greater amount when $\ell + n_i - n_d \geq 2(j + n_i) + 1$. Thus, we have

$$\Gamma(\ell, j, \beta) = \min_{n_i \in \mathbb{Z}^*} \left\{ n_i \mid \frac{m + n_i}{\ell + n_i} > \beta \right\},$$

if $\ell + i \geq 2(j + i) + 1$ for $1 \leq i < \Gamma(\ell, j, \beta)$. Also observe that $\frac{j+n_i}{\ell+n_i-n_d} > \beta$ implies $n_i > \frac{\beta\ell-j}{1-\beta}$. All that remains is to show that for $\beta \in (0, \frac{1}{2}]$ the constraint $\ell + i \geq 2(j + i) + 1$ is satisfied for $1 \leq i < \Gamma(\ell, j, \beta)$. To show this, we observe that if $i < \ell - 2j$, then $\ell + i \geq 2(j + i) + 1$. Thus, the constraint is satisfied if $\Gamma(\ell, j, \beta) \leq \ell - 2j$. Since $\frac{\beta\ell-j}{1-\beta} \leq \ell - 2j$ for all $\beta \in (0, \frac{1}{2}]$, we get the desired bound. \square

We can think of the variables n_i and n_d as the numbers of insertions and deletions into our data structure. Thus, $\Gamma(\ell, j, \beta)$ represents the number of updates that can occur in a range containing ℓ points before a colour c with j occurrences can possibly become a β -majority. We next prove the following lemma:

Lemma 8. *Suppose the list $L(v)$ for node v contains the $\lceil \frac{1-\beta+\sqrt{1-\beta}}{\beta} \rceil$ most frequent colours in the range $R(v)$, breaking ties arbitrarily. For $\beta \in (0, \frac{1}{2}]$, this value is upper bounded by $\lceil \frac{2}{\beta} \rceil$. Let ℓ be the number of points contained in $R(v)$. Only after $\lceil \frac{\beta\ell}{\sqrt{1-\beta}(1+\sqrt{1-\beta})} \rceil \geq \lceil \frac{\beta\ell}{2} \rceil$ insertions or deletions into $T(v)$ can a colour $c \notin L(v)$ possibly become a β -majority for the range spanned by node v .*

Proof. Since we keep in $L(v)$ the k most frequently appearing colours in the range $R(v)$, any colour not in $L(v)$ can appear at most $\frac{\ell}{k+1}$ times. We apply Lemma 7, noting that it exactly describes the number of insertions or deletions required to cause a colour with frequency m to become a β -majority in a range containing ℓ points. Thus, we get that $\Gamma(\ell, \frac{\ell}{k+1}, \beta) \geq \frac{\beta\ell - \ell/(k+1)}{1-\beta}$. We want to maximize the ratio $\Gamma(\ell, \frac{\ell}{k+1}, \beta)/k$, which gives us

the maximum number of updates before rebuilding $L(v)$ per colour stored in v . If $h(k) = \frac{\beta\ell - \ell/(k+1)}{(1-\beta)k}$, then the derivative $h'(k) = \frac{(-2k+\beta k^2+2\beta k+\beta-1)\ell}{(k+1)^2(-1+\beta)k^2}$, which has zeros at $k = \{\frac{1-\beta+\sqrt{1-\beta}}{\beta}, \frac{1-\beta-\sqrt{1-\beta}}{\beta}\}$. The relevant zero, which maximizes $h(k)$ is $k = \frac{1-\beta+\sqrt{1-\beta}}{\beta}$. Substituting this in as k into $\frac{\beta\ell - \ell/(k+1)}{1-\beta}$, we get that $\frac{\beta\ell}{\sqrt{1-\beta}(1+\sqrt{1-\beta})}$ updates are required before a colour $c \notin L(v)$ can become a β -majority for the range spanned by node v . \square

By Lemma 8, our lazy updating scheme only requires each list $L(v)$ to have size $\lceil \frac{1-\beta+\sqrt{1-\beta}}{\beta} \rceil = O(1/\alpha)$. This leads to the following theorem:

Theorem 1. *Given a set \mathcal{P} of n points in one dimension and a fixed $\alpha \in (0, 1)$, there is an $O(n)$ space data structure that supports range α -majority queries on \mathcal{P} in $O((\lg n)/\alpha)$ time, and insertions and deletions in $O((\lg n)/\alpha)$ amortized time.*

Proof. Query time follows from Lemma 6. In order to get the desired space, we combine Lemmas 6 and 8, implying that each list $L(v)$ contains $O(1/\alpha)$ colours. This allows us to use the same pruning technique described in Lemma 6 in order to reduce the space to $O(n)$.

When an update occurs, we follow the path from the root of T to the updated node u . Suppose, without loss of generality, the update is an insertion of a point of colour c . For each vertex v on the path, if v contains a list $L(v)$, we check whether c is in $L(v)$. If it is, then we increment the count of colour c . This takes $O(1/\alpha)$ time. We also increment the counter for v that keeps track of the number of updates into $T(v)$ that have occurred since $L(v)$ was rebuilt. Thus, modifying the lists and counters along the path requires $O((\lg n)/\alpha)$ time in the worst case.

We next look at the costs of maintaining the lists $L(v)$. The list $L(v)$ can be rebuilt in $O(|T(v)|)$ time, using the array D . Note that D can be maintained under updates using the same scheme described in Section 1.2. First, we use D to compute the frequency of all the colours in $R(v)$ in $\Theta(|T(v)|)$ time. Let k be the value from Lemma 8. Since there are at most $O(|T(v)|)$ colours, we can use a linear time selection algorithm to find the k -th most frequent colour in D , and then find the top k most frequent colours via a linear scan in $O(|T(v)|)$ time. We can then enforce the necessary semi-ordering on this list in $O(k) = O(\frac{1}{\alpha})$ time, as described in Section 2.1. Thus, each leaf in $T(v)$ pays $O(1)$ cost every $\Theta(|T(v)|\alpha)$ insertions, or $O(1/\alpha)$ amortized cost per insertion. Since each update may cause $O(\lg n)$ lists to be rebuilt, this increases the cost to $O((\lg n)/\alpha)$ amortized time per update.

We make use of standard local rebuilding techniques to keep the tree T balanced, rebuilding the lists in nodes that are merged or split during an update. Since a node v will only be merged or split after $O(|T(v)|)$ updates by the properties of weight-balanced B-trees, these local rebuilding operations require $O(\lg n/\alpha)$ amortized time. Finally, we can update F_c and F during an insertion or deletion of a point of colour c in $O(\lg n)$ time. Thus, updates require $O((\lg n)/\alpha)$ amortized time overall, and are dominated by the costs of maintaining the lists $L(v)$ in each node v . \square

3 Speedup for Integer Coordinates

We next describe how to improve the query time of the data structure from Theorem 1 from $O((\lg n)/\alpha)$ to $O(\lg n/(\alpha \lg \lg n))$ for the case in which the x -coordinates of the points in \mathcal{P} are integers that can be stored in a constant number of words.

To accomplish this goal, we require an improved one-dimensional range counting data structure, which we get by combining two existing data structures. The fusion tree of Fredman and Willard [15] is an $O(n)$ space data structure that supports predecessor and successor queries in $O(\lg n/\lg \lg n)$ time and insertions/deletions in $O(\lg n/\lg \lg n)$ amortized time. The list indexing data structure of Dietz [11] uses $O(n)$ space and supports *rank queries*—i.e., given an element, return the number of elements that precede it in the list—in $O(\lg n/\lg \lg n)$ time, and insertions/deletions in $O(\lg n/\lg \lg n)$ amortized time. In Andersson *et al.* [1], it was observed that these data structures could be combined to support dynamic one-dimensional range counting queries in $O(\lg n/\lg \lg n)$ time per operation; amortized for updates. We refer to this data structure as an *augmented fusion tree*.

In order to achieve $O(\lg n / (\alpha \lg \lg n))$ query time, we implement all the range counting data structures as augmented fusion trees: i.e., the data structures F , and F_c for each $c \in C$. Immediately, we get that we can perform a query in $O(\lg n / (\alpha \lg \lg n) + \lg n)$ time: $O(\lg n / (\alpha \lg \lg n))$ time for the range counting queries, and $O(\lg n)$ time to find the nodes in I_t . We now discuss how to remove the additive $O(\lg n)$ term, which involves modifying our weight-balanced B-tree to support dynamic lowest common ancestor queries. To identify the top $O(\lg \frac{1}{\alpha})$ levels of I , we use the following lemma:

Lemma 9. *The weight-balanced B-tree T can be augmented in order to support lowest common ancestor queries in $O(\sqrt{\lg n})$ time without changing the $O((\lg n / \alpha))$ amortized time required for updates.*

Proof. Let the first ancestor of a node u be the parent of u , and the ℓ -th ancestor of u be the parent of the $(\ell - 1)$ -th ancestor of u for $\ell > 1$. In order to support lowest common ancestor queries between two nodes z_a and z_b , denoted $\text{LCA}(z_a, z_b)$, we add three pointers to each node $u \in T$: pointers to both the leaves representing the minimum and maximum x -coordinates in $T(u)$, and a pointer to the ℓ -th ancestor of u ; we will fix the value of ℓ later. We can search for the $\text{LCA}(z_a, z_b)$ by setting $v = z_a$ and following the pointer to the ℓ -th ancestor of v , denoted v' . By checking the maximum x -coordinate to see if $R(v')$ contains z_b , we can determine whether v' is an ancestor of $\text{LCA}(z_a, z_b)$ or a descendant of $\text{LCA}(z_a, z_b)$ in constant time. If v' is a descendant of $\text{LCA}(z_a, z_b)$, then we set v to v' and v' to the ℓ -th ancestor of v' . If v' is an ancestor of $\text{LCA}(z_a, z_b)$, then we backtrack and walk up the path from v to v' until we find $\text{LCA}(z_a, z_b)$. Overall, it takes $O(h_0/\ell + \ell)$ time to find node $z = \text{LCA}(z_a, z_b)$, if z is at height h_0 in T . By setting $\ell = O(\sqrt{\lg n})$ we get $O(\sqrt{\lg n})$ time. Furthermore, the pointers we added to T can be updated in $O(\lg n)$ amortized time during an insertion or deletion. Whenever we merge or split a node u , we have $O(|T(u)|)$ time to fix all of the pointers into u , by properties of weight-balanced B-trees. The pointers out of u can be fixed in $O(\lg n)$ worst case time. \square

Although Lemma 9 is weaker than other results (cf. [29]), it is simple and sufficient for our needs. We next present the following theorem:

Theorem 2. *Given a set \mathcal{P} of n points in one dimension with integer coordinates and a fixed $\alpha \in (0, 1)$, there is an $O(n)$ space data structure that supports range α -majority queries on \mathcal{P} in $O(\lg n / (\alpha \lg \lg n))$ time, and both insertions and deletions into \mathcal{P} in $O((\lg n) / \alpha)$ amortized time.*

Proof. Suppose we are given a query range $[x_a, x_b]$. Applying Lemma 9 to the weight-balanced B-tree T , we claim that we can identify the top ℓ levels of I —that are *not* necessarily from consecutive levels in T —using $O(\ell)$ least common ancestor operations. To show this, we describe a recursive procedure $\text{FINDTOP}(z_a, z_b, \ell)$ for identifying the top ℓ levels of I . We assume that we have acquired pointers to z_a and z_b , the leaves of T that represent the x -coordinates of the successor of x_a and predecessor of x_b , respectively. To do this, we add a pointer from each leaf in the augmented fusion tree F to its corresponding leaf in T . Given a query, we initially perform a successor query for x_a and predecessor query for x_b in F , and follow these extra pointers to z_a and z_b , respectively. We assume that $z_a \neq z_b$, otherwise the query is trivially answered by reporting the colour stored in z_a .

Let $z = \text{LCA}(z_a, z_b)$, and c_i denote the i -th child of z . Let z_l and z_r denote the leftmost and rightmost leaves in T_z . In constant time we can determine children c_j and c_k of z which are on the path to z_a and z_b , respectively. Note that $k - j > 0$, otherwise z is not the $\text{LCA}(z_a, z_b)$. We say we are in the *good case* when $z_a = z_l$, $z_b = z_r$, and/or $k - j > 1$. When we are in the good case, either c_j , c_k , and/or c_{j+1}, \dots, c_{k-1} are in the top level of I , and we set $\ell' = \ell - 1$. Otherwise, if $k - j = 1$ and $z_a \neq z_l$ and $z_b \neq z_r$, then we are in the *bad case*. In the bad case we have not found the top level of I , and we set $\ell' = \ell$. In both cases (good or bad), let $z_{b'}$ be the leaf in c_k representing the minimum x -coordinate in T_{c_k} , and $z_{a'}$ be the leaf in c_j representing the maximum x -coordinate in T_{c_j} . We recurse if $\ell' > 0$, calling $\text{FINDTOP}(z_a, z_{a'}, \ell')$ if $z_a \neq z_{a'}$ and $\text{FINDTOP}(z_{b'}, z_b, \ell')$ if $z_b \neq z_{b'}$.

We observe that the procedure $\text{FINDTOP}(z_a, z_b, \ell)$ uses $O(\ell)$ least common ancestor queries. This is because if a call to FINDTOP is in the bad case, then the subsequent recursive call(s) will be in the good case by choice of $z_{a'}$ and $z_{b'}$, and only the initial call to FINDTOP can make two recursive calls. Using FINDTOP ,

we can identify the top $O(\lg \frac{1}{\alpha})$ levels of I in $O(\sqrt{\lg n} \times \lg \frac{1}{\alpha})$ time, replacing the $O(\lg n)$ additive term. This factor is strictly asymptotically less than the time required to perform the range-counting queries, which is $O(\lg n / (\alpha \lg \lg n))$.

By Lemma 9, we can support the lowest common ancestor operation without increasing the update time of T as stated in Theorem 1. The extra pointers we added from the leaves of F to the leaves of T can also be updated without affecting the bound from Theorem 1, since during any insertion/deletion of a point p , the two leaves corresponding to p in both F and T must be located. Therefore, the total update time follows from Theorem 1. \square

4 Extension to Higher Dimensional Point Sets

In this section we present a refinement of the technique presented by Karpinski and Nekrich [23], who used standard range tree techniques [4] to generalize their range α -majority structures to higher dimensions.⁵ We note that, recently, Wilkinson [31] has used the same refinement to improve the bounds of Durocher *et al.* [13] for the two-dimensional static case.

All of the algorithms presented thus far have the following two phase structure. The first is the *candidate extraction* phase, in which we extract a list of candidates from our data structure. The second phase is the *verification* phase, in which we use range counting data structures to verify that they are actual α -majorities. For higher dimensional problems we speed up the verification phase by adding an additional filtering phase between the candidate extraction and verification phases.

In order to do this, we make use of *approximate range counting* data structures [26,28,31]. If m points are contained in the query range, then an approximate range counting data structure with additive error m' will return a count in the range $[m - m', m + m']$; see [28]. Similarly, a data structure with multiplicative error $(1 - \varepsilon)$ will return a count in the range $[(1 - \varepsilon)m, m]$; see [26]. In the remainder of this section we first modify existing data structures for approximate range counting, and then consider their applications to higher dimensional data structures for dynamic range α -majority queries.

4.1 Approximate Range Counting

Before stating the results for higher dimensional range α -majority data structures we require some additional results on approximate range counting. We begin with a lemma, which is a very minor generalization of Nekrich's one-dimensional approximate range counting data structure [28, Theorem 1]. In the original structure each point is unweighted, but we wish to add the operations **increment** and **decrement** to the structure, which respectively increase and decrease the weight of a point by one. We assume a newly inserted point begins with weight one. Instead of returning the number of points in a query range (within an additive error term), our query operation will return the sum of the weights of the points in the query range, within an additive error term.

Lemma 10. *Let $\tau > 1$ be an integer constant, $\mathbf{dpred}(n)$ denote the cost of a dynamic predecessor search on n keys, and m denote the sum of the weight of the points contained the query range. There exists an $O(n)$ space data structure that supports approximate weighted range counting queries with additive error $m^{1/\tau}$ in $O(\mathbf{dpred}(n) + \lg \lg n)$ time, deletions in $O(\lg \lg n)$ amortized time, and insertions in $O(\mathbf{dpred}(n) + \lg \lg n)$ amortized time. The operations **increment** and **decrement** are supported in $O(\lg \lg n)$ amortized time.*

Proof. Let m' be the approximate weight returned by our data structure, while m is the exact weight. We divide the solution into two cases. In the first case, we assume that $m \geq h_0(\lg n)^\tau$ for some arbitrary constant $h_0 > 0$. We emphasize that in both cases the data structure and proof are essentially the same as Theorem 1 of Nekrich [28], with some minor modifications. We use an exponential search tree T [2], where each leaf in T represents a point, but also stores the weight associated with the point. We require some additional

⁵ In the preliminary version of this paper [14], the paragraph preceding Theorem 3 erroneously stated that the result which we prove in this section follows immediately from the analysis of Karpinski and Nekrich [23].

notation, and closely follow that of Nekrich [28]. Let v_i denote the i -th child of v , n_v denote the number of leaves in the subtree $T(v)$, $W(v)$ denote the weight of the leaves of the subtree $T(v)$, and $f(v)$ denote the number of children of v . In the exponential search tree T , each node v has $\Theta(n_v^{1/\tau})$ children, each of which contains between $n_v^{(\tau-1)/\tau}/2$ and $2n_v^{(\tau-1)/\tau}$ points, for a fixed constant $\tau > 2$. Each node v stores its weight $W(v)$, as well as a set of approximate weights $W'(v, i, j)$, such that

$$W(v_i) + \dots + W(v_j) - n_v^{3/\tau}/2 \leq W'(v, i, j) \leq W(v_i) + \dots + W(v_j) + n_v^{3/\tau}/2 ,$$

for all $1 \leq i \leq j \leq f(v)$. We recompute all counts $W'(v, i, j)$ after $n_v^{3/\tau}/2$ update operations (insertions, deletions, increments, and decrements). Recomputing all the $W'(v, i, j)$'s for a node takes $O(n_v^{2/\tau})$ time. Thus, each update operation—insertion, deletion, increment, decrement—requires $O(\lg \lg n)$ amortized time, since the height of T is $O(\lg \lg n)$ and we must increment or decrement the weight $W(v)$ stored in each node on the path from leaf to root.

The space is linear by the properties of exponential search trees [2], and all that remains is to argue the correctness of the query algorithm of Nekrich [28]. The query algorithm essentially finds the ranges in T that represent the query range, and returns the summation of the approximate counters of those ranges.

For a fixed node v from the set of nodes representing the query range, with children v_i, \dots, v_j contained entirely in the query range, let $m'_v = W'(v, i, j)$ and $m_v = \sum_{\ell=i}^j W(v_\ell)$. Then $m_v - n_v^{3/\tau} \leq m'_v \leq m_v + n_v^{3/\tau}$. Since $m_v \geq n_v^{(\tau-1)/\tau}$, $m_v - m_v^{3/(\tau-1)} \leq m'_v \leq m_v + m_v^{3/(\tau-1)}$. Since T has height $h_1 \lg \lg n$ for some constant h_1 , $m - (2h_1 \lg \lg n)(m^{3/(\tau-1)}) \leq m' \leq m + (2h_1 \lg \lg n)(m^{3/(\tau-1)})$. However, since we assume $m \geq h_0(\lg n)^\tau$, we need only ensure $h_0 > (2h_1)^\tau$ in order for $2h_1 \lg \lg n \leq m^{1/(\tau-1)}$. Thus, $m - m^{4/(\tau-1)} \leq m' \leq m + m^{4/(\tau-1)}$. By replacing τ with $\tau' = \max(5\tau, 5)$ we obtain the result of the lemma.

In the second case, when $m < h_0(\lg n)^\tau$, we make use of an alternative data structure. We divide the point set into groups of between $h_0(\lg n)^\tau$ and $4h_0(\lg n)^\tau$ consecutive points, and store each group in a balanced binary search tree. Each node u in the search tree stores the total weight of the nodes in the subtree induced by u . Given the successor and predecessor, e'_1 and e'_2 , of a query range $[e_1, e_2]$, we can assume that points e'_1 and e'_2 either belong to the same group, or two adjacent groups. Thus, given e'_1 and e'_2 , which can be obtained in $O(\text{dpred}(n))$ time, we can tally the *exact* weight in this case in $O(\lg \lg n)$ time. Using standard techniques we can support insertion in $O(\text{dpred}(n) + \lg \lg n)$ amortized time; $O(\text{dpred}(n))$ to find the position in which to insert the new element, and $O(\lg \lg n)$ amortized time to insert it into the binary search tree for its group, accounting for merging/splitting of groups. By analogous arguments deletion takes $O(\lg \lg n)$ amortized time. Finally, increment and decrement can be performed in $O(\lg \lg n)$ worst case time. \square

Before continuing, we require the definition of a generalized union-split-find (GUSF) data structure, as well as the time bounds for its operations.

Lemma 11 ([19], Theorem 5.2). *A GUSF stores an ordered list G of elements, in which each element x of G is associated with a subset $U(x) \subseteq \{1, \dots, \lg^{\frac{1}{4}} n\}$ of colours. Assume we have a pointer to an element $x \in G$, and $U' \subseteq \{1, \dots, \lg^{\frac{1}{4}} n\}$ be a set of colours. A GUSF supports the operations:*

- *$\text{find}(x, U')$: return the successor of x with colour $c \in U'$.*
- *$\text{add}(y, x)$: inserts y into the list before x .*
- *$\text{erase}(x)$: removes x from the list, assuming $U(x) = \emptyset$.*
- *$\text{mark}(x, c)$: inserts c into $U(x)$.*
- *$\text{unmark}(x, c)$ removes c from $U(x)$.*

A GUSF can be implemented in $O(n)$ space, such that each operation takes $O(\lg \lg n)$ time. The time bound for add and erase is amortized, while the running time of all other operations are worst case. A GUSF containing n elements can be constructed in $O(n \lg^{\frac{1}{8}} n \lg \lg n)$ time.

Next we are ready to state and prove the main result of this section:

Lemma 12. *Let \mathcal{P} be a set of d -dimensional points for any $d \geq 2$. The point set, \mathcal{P} , can be preprocessed into an $O((n \lg^{d-1} n) / \lg \lg n)$ space data structure, such that for any arbitrary d -dimensional axis-aligned hyperrectangle, \mathcal{Q} , approximate range counting queries can be performed in $O(\lg^{d-1} n)$ time, with additive error $|\mathcal{P} \cap \mathcal{Q}|^{\frac{1}{j}}$, for any constant integer $j > 1$. Insertions and deletions can be performed in $O(\lg^{d-1+\frac{1}{4}} n)$ amortized time.*

Proof. The proof of this lemma is *almost* the same as Theorem 2 in [28], except that we increase the cost of the query by a factor of $O(\lg \lg n)$ for any $d \geq 3$, and decrease the space bound by a factor of $O(\lg^\epsilon n)$. We make use of dynamic fractional cascading in weight balanced B-trees without modifications [19], and a slight modification of the GUSF of Lemma 11.

We next describe how to combine the one-dimensional weighted approximate counting data structure from Lemma 10 with the GUSF. This will allow the GUSF to support coloured approximate range counting: i.e., given a colour $c \in \{1, \dots, \lg^{\frac{1}{4}} n\}$ and a range $[x_a, x_b]$, approximately report the number of elements with colour c contained in $[x_a, x_b]$. A single *modified* GUSF will then be stored in each internal node of a weight-balanced B-tree, in order to support two-dimensional approximate range counting.

A GUSF groups consecutive elements into *blocks* which are of size $\Theta(\lg^{2+\frac{1}{4}} n)$. The elements in each block are stored in a balanced binary search tree. For each node in the tree, we store the counts of the number of children with each of the $\lg^{\frac{1}{4}} n$ colours, with counter n_c storing the number of points of colour c . Since the tree has $O(\lg^{2+\frac{1}{4}} n)$ elements, each counter requires $O(\lg \lg n)$ bits, and thus the counters for a node can be packed into a constant number of words. Thus, these counters do not increase the space of the GUSF structure asymptotically.

As in a standard GUSF, each block in the modified GUSF is represented in an order maintenance structure that maps a block to an integer coordinate. Given two blocks, b and b' , we denote their corresponding integer coordinates $X(b)$ and $X(b')$, and we can determine whether the elements in b precede those in b' , or vice versa, by comparing these coordinates; see [19] for full details.

Our modified GUSF also stores $O(\lg^{\frac{1}{4}} n)$ copies of the data structure from Lemma 10, one for each colour $c \in [1, \lg^{\frac{1}{4}} n]$, denoted D_c . We discuss how to set the parameter τ for each D_c later. For each block b that has a counter value $n_c > 0$ in its root for colour c , we store a point representing that block in D_c , with weight n_c , and coordinate $X(b)$. The root of b also stores a pointer to the leaf in D_c representing these points, for each colour $c \in [1, \lg^{\frac{1}{4}} n]$. Since there are at most $O(n / \lg^{2+\frac{1}{4}} n)$ blocks, all these structures together occupy $O(n / \lg^2 n)$ space.

Given two elements e_1 and e_2 , where $e_1 < e_2$ and both elements are marked with colour c , we can determine the approximate number of points with colour c that both succeed e_1 and precede e_2 , as follows. First, if e_1 and e_2 are in the same block, we can return the *exact count* in $O(\lg \lg n)$ time using the counters that are stored in the nodes of the balanced binary tree representing the block. Otherwise, we need to perform an additional step of querying the data structure D_c , providing pointers to the leaves in D_c that represent the blocks containing e_1 and e_2 , respectively.

With the exception of the data structures D_c , the GUSF containing n elements can be constructed in $O(n \lg^{\frac{1}{8}} n \lg \lg n)$ time by Lemma 11, since each GUSF operation takes at most $O(\lg \lg n)$ amortized time. Since each point results in an insertion or increment operation on $O(\lg^{\frac{1}{8}} n)$ approximate range counting data structures, each of size $O(n / \lg^{2+\frac{1}{4}} n)$, this takes $O(\lg^{\frac{1}{8}} n \lg \lg n)$ amortized time per point, and does not asymptotically change the construction time.

We are next ready to discuss our data structure for planar approximate counting, i.e., the case in which $d = 2$. We store a weight balanced B-tree T over the y -coordinates of the given points, with branching parameter $\Theta(\lg^{\frac{1}{8}} n)$ and leaf parameter 1. For each internal node u of T with degree f , we store our modified GUSF $M(u)$, over all of the points in the subtree $T(u)$, ordered by their x -coordinate. Note that there are $\Theta(\lg^{\frac{1}{4}} n)$ possible contiguous subranges of children of u in total, and each child of u belongs to $\Theta(\lg^{\frac{1}{8}} n)$ of these ranges $[i, j]$, where $1 \leq i \leq j \leq f$. We construct a set of colours, and each colour corresponds to a possible range $[i, j]$. Thus, each point in $M(u)$ is marked with the $\Theta(\lg^{\frac{1}{8}} n)$ colours corresponding to these ranges. Each node u also stores a catalogue $V(u)$ corresponding to the points in $T(u)$ ordered by

x -coordinate. Each catalogue element stores a pointer to the corresponding element in $M(u)$. We maintain a dynamic fractional cascading data structure over the catalogues of T .

Since the branching parameter of the tree T is $\Theta(\lg^{\frac{1}{8}} n)$, the tree has height $\Theta(\lg n / \lg \lg n)$. Each point is stored in $\Theta(\lg n / \lg \lg n)$ nodes, each containing a constant number of linear space structures. Thus, the space occupied by the data structure is $\Theta(n \lg n / \lg \lg n)$.

To answer a query of the form $[x_1, x_2] \times [y_1, y_2]$, we perform a search for the successor and predecessor, e_1 and e_2 , of the query range $[x_1, x_2]$ in each catalogue of each node among the nodes representing $[y_1, y_2]$ in T . This takes $\Theta(\lg n)$ time, since there are $\Theta(\lg n / \lg \lg n)$ catalogues: the initial search requires $\Theta(\lg n)$ time, and each additional search uses $\Theta(\lg \lg n)$ time. For a fixed internal node u of T , such that the query range $[y_1, y_2]$ spans children $[i, j]$, let c be the colour representing $[i, j]$ in $M(u)$. We locate e'_1 and e'_2 , the respective successor and predecessor of e_1 and e_2 in $M(u)$ with colour c , using the **find** operation. Thus, locating e'_1 and e'_2 in $M(u)$ takes $O(\lg \lg n)$ time. Once we have located e'_1 and e'_2 we can query D_c , and the counters in the block(s) containing e'_1 and e'_2 , in $O(\lg \lg n)$ time, as outlined above. Thus, the overall query time is $O(\lg \lg n (\lg n / \lg \lg n) + \lg n)$ which is $O(\lg n)$.

Suppose we desire additive error $|\mathcal{P} \cap \mathcal{Q}|^{\frac{1}{j}}$ for some fixed constant $j > 1$. Then, we set the parameter $\tau = 2j$. Let m' denote the sum of each of the $h_2(\lg n / \lg \lg n)$ approximate counts tallied at each node that represents the query range, where h_2 is a constant that depends on the height of T , and m denotes the exact count. Thus,

$$m - h_2(\lg n / \lg \lg n)m^{\frac{1}{\tau}} \leq m' \leq m + h_2(\lg n / \lg \lg n)m^{\frac{1}{\tau}}. \quad (1)$$

If $m \geq (h_2 \lg n)^\tau$, then $m^{\frac{1}{\tau}} \geq h_2(\lg n / \lg \lg n)$. Thus, $m - m^{\frac{2}{\tau}} \leq m' \leq m + m^{\frac{2}{\tau}}$. Since $\tau = 2j$, we are left with $m - m^{\frac{1}{j}} \leq m' \leq m + m^{\frac{1}{j}}$, which is the desired error term.

Next suppose $m < (h_2 \lg n)^\tau$. In this case, we can retrieve the exact count in $O(\lg n)$ time using the binary tree representation of D_c , since none of the ranges represented can contain more than m points. Thus, in both cases we have shown the query algorithm is correct. Note that we must ensure $h_0 \geq h_2^\tau$, in addition to the constraints on h_0 described in Lemma 10.

In order to insert a point p , we identify the nodes on the path Y from the root of T to the leaf where p will be inserted. We then search for the successors of p in all of the catalogues on this path, which takes $O(\lg n)$ time in total. Once we have the successor, we can insert p into each GUSF along Y in the following way. Let u be a node in Y and u_i be the child of u whose range contains p . Using the pointer to the successor of p in $M(u)$, we can perform an **add** operation, inserting p into a block b in $M(u)$. Let U' denote the set of colours in $M(u)$ representing the ranges that contain u_i .

If b splits into two blocks b and b' as a result of this, then we must decrement the weight of the element representing b in each D_c for each c with a non zero counter in the root of b . We also must insert a new element representing b' into each D_c for each c with a non zero counter in the root of b' , and increment its weight accordingly. Recall that $O(\lg^{2+\frac{1}{4}} n)$ elements must have been inserted into $M(u)$ to cause b to split. Each split causes $O(\lg^{\frac{1}{8}} n (\lg^{2+\frac{1}{4}} n))$ update operations on all D_c , for each c stored in the roots of b and b' . This is $O(\lg^{\frac{1}{8}} n \lg \lg n)$ amortized update time. In the case in which b does not split, we still require $O(\lg^{\frac{1}{8}} n \lg \lg n)$ amortized time to **increase** the weight of the representative of b in each D_c , for $c \in U'$. Since there are $O(\lg n / \lg \lg n)$ nodes in Y , the overall insertion time is thus $O(\lg^{1+\frac{1}{8}} n)$, provided we do not cause a node to split or two nodes to merge in the base tree T . Deletion is handled analogously, except that we **decrease** the weight of the representative of b in each D_c for $c \in U'$. Thus, deletion requires $O(\lg^{1+\frac{1}{8}} n)$ amortized time as well.

In the case in which a node $u \in T$ splits or merges, we can efficiently update the fractional cascading data structure using the techniques described in [19]. The cost of a split or merge is dominated by the cost of rebuilding the modified GUSF structures in both u and u 's parent. We can rebuild each modified GUSF in a node representing m points in $O(m \lg^{\frac{1}{8}} n \lg \lg n)$ time. Since $O(m)$ updates are required to split a node with a parent containing $O(m \lg^{\frac{1}{8}} n)$ points, and $O(\lg n / \lg \lg n)$ splits/merges can occur during a single update, the cost of performing an update is $O(\lg^{1+\frac{1}{4}} n)$ amortized time.

To get the bound stated by the lemma for higher dimensions, we use the standard range tree technique [4], which inflates the space, query and update time by a factor of $O(\lg n)$ for each additional dimension. In general, we must set the parameter $\tau = 2^{d-1}j$. \square

4.2 Range α -Majority in Higher Dimensions

As an application of Theorem 1, and the approximate range counting data structures of Lemma 12, we can improve the query time for range α -majority data structures in higher dimensions.

Theorem 3. *Given a set \mathcal{P} of n points in d dimensions (for a constant $d \geq 2$) and a fixed $\alpha \in (0, 1)$, there is an $O(n \lg^{d-1} n)$ space data structure that supports range α -majority queries on \mathcal{P} in $O((\lg^d n)/\alpha)$ time, and insertions and deletions into \mathcal{P} in $O((\lg^d n)/\alpha)$ amortized time.*

Proof. Using range trees, we can convert any d -dimensional range α -majority query into a combination of several $(d-1)$ -dimensional range α -majority queries and d -dimensional range counting queries. In particular, let $\mathcal{S}(n, d)$ denote the cost of a d -dimensional range counting query on a dynamic set of n points, and $\mathcal{M}(n, d)$ denote the cost of a d -dimensional range α -majority on a dynamic set of n points. Then, for $d \geq 2$,

$$\mathcal{M}(n, d) = O(\lg n)\mathcal{M}(n, d-1) + O(\lg n/\alpha)\mathcal{S}(n, d) , \quad (2)$$

since we can extract and verify the frequency of the $O((\lg n)/\alpha)$ candidates from the $O(\lg n)$ nodes representing the range spanned by the d -th coordinate of the query range. Note that each candidate is an α -majority if we consider their first $(d-1)$ coordinates only. Since d -dimensional dynamic orthogonal range counting queries require $O(\lg^d n)$ time [8], $\mathcal{M}(n, d) = O((\lg^{d+1} n)/\alpha)$.

To further reduce the query time we observe that only $O(1/\alpha)$ of the $O(\lg n/\alpha)$ candidates can have frequency above $(1-\varepsilon)\alpha q$, where q is the total number of points contained in the query range and $\varepsilon \in (0, 1)$ is an arbitrary constant. Thus, we add additional data structures F'_c for each $c \in C$, where each F'_c is the structure of Lemma 12 and stores the points of colour c .⁶ Using these data structures, we can perform an additional filtering pass of the list of $O(\lg n/\alpha)$ candidates into a shorter list of $O(1/\alpha)$ candidates. After this filtering step we can then verify the frequency of the $O(1/\alpha)$ candidates above this threshold exactly using range counting data structures F_c . Let $\hat{\mathcal{S}}(n, d)$ denote the query time of the data structure from Lemma 12. We can rewrite the recurrence of Equation 2 as:

$$\mathcal{M}(n, d) = O(\lg n)\mathcal{M}(n, d-1) + O(\lg n/\alpha)\hat{\mathcal{S}}(n, d) + O(1/\alpha)\mathcal{S}(n, d) . \quad (3)$$

This recurrence resolves to $\mathcal{M}(n, d) = O((\lg^d n)/\alpha)$. We can update the structures F_c and F'_c for the colour, c , of the inserted or deleted point, and F in $O(\lg^d n + \lg^{d-1+\frac{1}{4}} n)$ amortized time. Each of the $O(\lg n)$ $(d-1)$ -dimensional range α -majority data structures can be updated in $O((\lg^{d-1} n)/\alpha)$ amortized time, for a total of $O((\lg^d n)/\alpha)$ amortized time. Finally, the space cost is dominated by the range α -majority structures. The space occupied by this can be expressed as $\mathcal{U}(n, d) = O(\lg n)\mathcal{U}(n, d-1)$, where $\mathcal{U}(n, d)$ is the space occupied by a d -dimensional dynamic range α -majority structure, and $d \geq 2$. Thus $\mathcal{U}(n, d) = O(n \lg^{d-1} n)$. \square

5 Dynamic Arrays

In this section we extend our results to dynamic arrays. In the dynamic array version of the problem, we wish to support the following operations on an array A of length n , where each $A[i]$ contains a colour, for $1 \leq i \leq n$:

- INSERT(i, c): Insert the colour c between the colours $A[i-1]$ and $A[i]$. This shifts the colours in positions i to n to positions $i+1$ to $n+1$, respectively.

⁶ The data structure of Lemma 12 has very small additive error, though, for the purposes of this proof, we need only constant multiplicative error.

- **DELETE**(i): Delete the colour $A[i]$. This shifts the colours in positions $i + 1$ to n to positions i to $n - 1$, respectively.
- **MODIFY**(i, c): Set the colour $A[i]$ to c .
- **QUERY**(i, j): Let $|A[i..j]|_c$ denote the number of occurrences of colour c in the range $A[i..j]$. Report the set of colours M such that for each $c \in M$, $|A[i..j]|_c > \alpha|j - i + 1|$. As before, we refer to a colour $c \in M$ as an α -majority in the range $A[i..j]$, and the query as a *range α -majority query*.

The dynamic array problem boils down to the well-studied problem of maintaining an injective order preserving mapping from the positions in A into a larger set of integer keys \mathcal{P} [27]. We next prove the following theorem:

Theorem 4. *Given an array $A[1..n]$ of colours and a fixed $\alpha \in (0, 1)$, there is an $O(n)$ space data structure that supports range α -majority queries on A in $O((\lg n)/(\alpha \lg \lg n))$ time, **INSERT** in $O((\lg^3 n)/(\alpha \lg \lg n))$ amortized time, **DELETE** in $O((\lg n)/\alpha)$ amortized time, and **MODIFY** in $O((\lg n)/\alpha)$ amortized time.*

Proof. We maintain our data structure T from Theorem 2 on the integer key set \mathcal{P} . Each time a key p in \mathcal{P} is changed to key p' , we must delete p from T , and then insert p' into T . If an insertion or deletion into our dynamic array changes ℓ keys in the mapping, it will require $O((\ell \lg n)/\alpha)$ amortized time to change these values in T . We note that a **MODIFY** operation corresponds to one deletion and one insertion into T , requiring $O((\lg n)/\alpha)$ time.

We apply the *dynamic reduction to extended rank space* technique [27], which maps the positions in A to integers in the bounded universe $[1..O(n^3)]$. This mapping requires $O((\lg^2 n)/\lg \lg n)$ amortized time for insertions, and $O(1)$ amortized time for deletions. These time bounds also bound the number of key changes for insertion and deletion (in the amortized sense), completing the proof. \square

6 Conclusions

We have presented several new dynamic data structures for the range α -majority problem. These data structures improve on the previous results in terms of space, query, and update time.

Notably, for one-dimensional points, we presented a linear space data structure with $O(\lg n/\alpha)$ query time, and $O(\lg n/\alpha)$ amortized update time. In the case in which the coordinates of the points are integers, we reduced the query time by a $(\lg \lg n)$ -factor. This improved query time matches an existing cell-probe lower bound, for the case when $1/\alpha$ is a constant, and the word size is $\Theta(\lg n)$.

We also extended our one-dimensional data structure to d -dimensions, where $d \geq 2$ is an arbitrary constant. The generalized structure occupies $O(n \lg^{d-1} n)$ space, has $O(\lg^d n/\alpha)$ query time, and supports updates in $O(\lg^d n/\alpha)$ amortized time. It would be interesting to determine if either the space or query time can be improved for the higher dimensional data structure.

References

1. Andersson, A., Miltersen, P., Thorup, M.: Fusion trees can be implemented with AC^0 instructions only. *Theoretical Computer Science* 215(1-2), 337–344 (1999)
2. Andersson, A., Thorup, M.: Dynamic ordered sets with exponential search trees. *Journal of the ACM* 54 (June 2007)
3. Arge, L., Vitter, J.S.: Optimal external memory interval management. *SIAM Journal on Computing* 32(6), 1488–1508 (2003)
4. Bentley, J.: Multidimensional divide-and-conquer. *Communications of the ACM* 23(4), 214–229 (1980)
5. Blum, M., Floyd, R., Pratt, V., Rivest, R., Tarjan, R.: Time bounds for selection. *Journal of Computer and System Sciences* 7(4), 448–461 (1973)
6. Boyer, R.S., Moore, J.S.: MJRTY - A fast majority vote algorithm. In: Boyer, R.S. (ed.) *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pp. 105–117. *Automated Reasoning Series*, Kluwer, Dordrecht, The Netherlands (1991)

7. Bozanis, P., Kitsios, N., Makris, C., Tsakalidis, A.: New upper bounds for generalized intersection searching problems. In: Proceedings of the 22nd International Colloquium Automata, Languages and Programming. LNCS, vol. 944, pp. 464–474. Springer (1995)
8. Chazelle, B.: Functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing* 17(3), 427–462 (1988)
9. De Berg, M., Haverkort, H.: Significant-presence range queries in categorical data. In: Proceedings of the 8th International Workshop on Algorithms and Data Structures. LNCS, vol. 2748, pp. 462–473. Springer (2003)
10. Demaine, E., López-Ortiz, A., Munro, J.I.: Frequency estimation of internet packet streams with limited space. In: Proceedings of the 10th European Symposium on Algorithms. LNCS, vol. 2461, pp. 11–20. Springer (2002)
11. Dietz, P.: Optimal algorithms for list indexing and subset rank. In: Dehne, F., Sack, J., Santoro, N. (eds.) *Algorithms and Data Structures*, LNCS, vol. 382, pp. 39–46. Springer (1989)
12. Durocher, S., He, M., Munro, J.I., Nicholson, P.K., Skala, M.: Range majority in constant time and linear space. In: Proceedings of the 38th International Colloquium on Automata, Languages, and Programming. LNCS, vol. 6755, pp. 244–255. Springer (2011)
13. Durocher, S., He, M., Munro, J.I., Nicholson, P.K., Skala, M.: Range majority in constant time and linear space. *Information and Computation* (2012), to appear.
14. Elmasry, A., He, M., Munro, J.I., Nicholson, P.K.: Dynamic range majority data structures. In: Proceedings of the 22nd International Symposium on Algorithms and Computation. LNCS, vol. 7074, pp. 150–159 (2011)
15. Fredman, M., Willard, D.: Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences* 47(3), 424–436 (1993)
16. Gagie, T., He, M., Munro, J.I., Nicholson, P.K.: Finding Frequent Elements in Compressed 2D Arrays and Strings. In: Proceedings of the 18th Symposium on String Processing and Information Retrieval, LNCS, vol. 7024. Springer (2011)
17. Gagie, T., Kärkkäinen, J.: Counting colours in compressed strings. In: Proceedings of the 22nd Annual Symposium on Combinatorial Pattern Matching. LNCS, vol. 6661, pp. 197–207. Springer (2011)
18. Gagie, T., Navarro, G., Puglisi, S.: Colored range queries and document retrieval. In: Chavez, E., Lonardi, S. (eds.) *String Processing and Information Retrieval*, LNCS, vol. 6393, pp. 67–81. Springer (2010)
19. Giora, Y., Kaplan, H.: Optimal dynamic vertical ray shooting in rectilinear planar subdivisions. *ACM Transactions on Algorithms* 5, 28:1–28:51 (July 2009)
20. Gupta, P., Janardan, R., Smid, M.: Further Results on Generalized Intersection Searching Problems: Counting, Reporting, and Dynamization. *Journal of Algorithms* 19(2), 282–317 (1995)
21. Husfeldt, T., Rauhe, T.: New lower bound techniques for dynamic partial sums and related problems. *SIAM Journal on Computing* 32(3), 736–753 (2003)
22. Karp, R., Shenker, S., Papadimitriou, C.: A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems* 28(1), 51–55 (2003)
23. Karpinski, M., Nekrich, Y.: Searching for frequent colors in rectangles. In: Proceedings of the 20th Canadian Conference on Computational Geometry. pp. 11–14 (2008)
24. Lai, Y., Poon, C., Shi, B.: Approximate colored range and point enclosure queries. *Journal of Discrete Algorithms* 6(3), 420–432 (2008)
25. Misra, J., Gries, D.: Finding repeated elements. *Science of Computer Programming* 2(2), 143–152 (1982)
26. Mortensen, C.W.: *Data Structures for Orthogonal Intersection Searching and Other Problems*. Ph.D. thesis, IT University of Copenhagen (2006)
27. Nekrich, Y.: Space efficient dynamic orthogonal range reporting. *Algorithmica* 49(2), 94–108 (2007)
28. Nekrich, Y.: Data structures for approximate orthogonal range counting. In: Proceedings of the 20th International Symposium on Algorithms and Computation. LNCS, vol. 5878, pp. 183–192. Springer (2009)
29. Tsakalidis, A.: The nearest common ancestor in a dynamic tree. *Acta Informatica* 25(1), 37–54 (1988)
30. Wei, Z., Yi, K.: Beyond simple aggregates: indexing for summary queries. In: Proceedings of the 2011 ACM SIGMOD/PODS Conference. pp. 117–128. ACM (2011)
31. Wilkinson, B.: *Adaptive Range Counting and Other Frequency-Based Range Query Problems*. Ph.D. thesis, University of Waterloo (2012)